

1. 上课约定须知

2. 本次内容大纲

3. 详细课堂内容

- 3.1. Flink RPC 详解
- 3.2. 阅读源码的准备
- 3.3. Flink 集群启动脚本分析
- 3.4. Flink 主节点 JobManager 启动分析
- 3.5. WebMonitorEndpoint 启动和初始化源码剖析
- 3.6. ResourceManager 启动和初始化源码剖析
- 3.7. Dispatcher 启动和初始化源码剖

4. 本次课程总结

5. 本次课程作业

1. 上课约定须知

课程主题：Flink 源码解析 -- 第一次课（集群启动）

上课时间：20:00 - 23:00

课件休息：21:30 左右 休息10分钟

课前签到：如果能听见音乐，能看到画面，请在直播间扣 666 签到

2. 本次内容大纲

FLink 源码分析大概会有 4-5 次课：

- 1、RPC 分析
- 2、启动的 shell 脚本分析
- 3、集群启动 JobManager 启动分析
- 4、集群启动 TaskManager 启动分析
- 5、Flink Job 的构建和提交（三层图架构）
- 6、Flink Job具体提交（client如何提交的）
- 7、Flink Job具体提交（server端是如何处理）
- 8、申请 slot 是怎么做的 + slot的管理（申请 + 释放）
- 9、StreamTask 的执行
- 10、MailBox 线程模型
- 11、Flink 状态管理
- 12、Flink Checkpoint 机制

今天是 Flink 源码的第一次课程，主要讲解的是 Flink 的集群启动，主要内容分为以下六点：

- 1、Flink RPC 剖析
- 2、Flink 集群启动脚本分析
- 3、Flink 集群启动 JobManager 启动源码剖析
- 4、WebMonitorEndpoint 启动和初始化源码剖析
- 5、ResourceManager 启动和初始化源码剖析
- 6、Dispatcher 启动和初始化源码剖析

3. 详细课堂内容

3.1. Flink RPC 详解

学到现在，看了很多技术组件的源码，大致总结一下各大组件 RPC 实现：

技术组件	RPC 实现
HDFS	Netty
HBase	HBase-2.x 以前：NIO + ProtoBuf HBase-2.x 以后：Netty
ZooKeeper	BIO + NIO + Netty
Spark	Spark-1.x 基于 Akka Spark-2.x 基于 Netty
Flink	Akka + Netty

总结：Flink 的 RPC 实现：基于 Scala 的网络编程库：Akka

几个储备知识：

- 1、在 Spark 第一节源码课中，讲解了一个使用 Akka 来模拟实现 YARN 的程序。
- 2、ActorSystem 和 Actor/ActorRef 的概念，工作方式，通信方式等，注意 异步 和 mailbox 这两个概念。

类似的概念：

- 1、HDFS : proxy
- 2、Akka : ActorRef
- 3、Flink : XXXGateway

在阅读 Flink 源码过程中，如果你见到有这种类型的代码，其实就是在发送 RPC 请求

```
// resourceManagerGateway 就可以理解成：当前节点中，对于 ResourceManager 代理对象的封装
resourceManagerGateway.requestSlot();
// 代码跳转到：resourceManager.requestSlot();
```

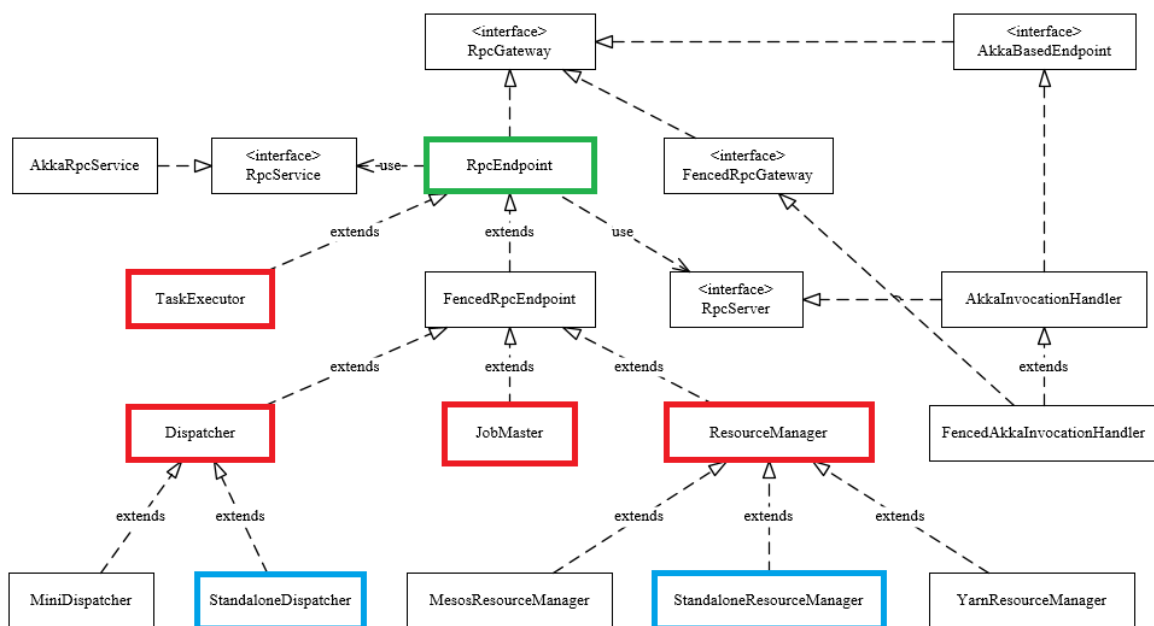
总结：Flink 中的 Endpoint，在作用上，等同于 Akka 中的 Actor

总结一下：

- 1、ActorSystem 是管理 Actor 生命周期的组件，Actor 是负责进行通信的组件
- 2、每个 Actor 都有一个 MailBox，别的 Actor 发送给它的消息都首先储存在 MailBox 中，通过这种方式可以实现异步通信。
- 3、每个 Actor 是单线程的处理方式，不断的从 MailBox 拉取消息执行处理，所以对于 Actor 的消息处理，不适合调用会阻塞的处理方法。
- 4、Actor 可以改变他自身的状态，可以接收消息，也可以发送消息，还可以生成新的 Actor
- 5、每一个ActorSystem 和 Actor都在启动的时候会给定一个 name，如果要从ActorSystem中，获取一个 Actor，则通过以下的方式进行 Actor的获取：
akka.tcp://asname@bigdata02:9527/user/actorname
- 6、如果一个 Actor 要和另外一个 Actor进行通信，则必须先获取对方 Actor 的 ActorRef 对象，然后通过该对象发送消息即可。
- 7、通过 tell 发送异步消息，不接收响应，通过 ask 发送异步消息，得到 Future 返回，通过异步回到返回处理结果。

Flink 中的 RPC 实现主要在 flink-runtime 模块下的 org.apache.flink.runtime.rpc 包中，涉及到的最重要的 API 主要是以下这四个：

- | | |
|---------------|---|
| 1、RpcGateway | 路由，RPC的老祖宗，各种其他RPC组件，都是 RpcGateway 的子类 |
| 2、RpcServer | RpcService (ActorSystem) 和 RpcEndpoint (Actor) 之间的粘合层 |
| 3、RpcEndpoint | 业务逻辑载体，对应的 Actor 的封装 |
| 4、RpcService | 对应 ActorSystem 的封装 |



最终总结一下：RpcEndpoint 下面有四个比较重要的子类：

- 1、TaskExecutor 集群中从节点中最重要的角色，负责资源管理
- 2、Dispatcher 主节点中的一个工作角色，负责 job 调度执行
- 3、JobMaster 应用程序中的 主控程序，类似于 spark 中的Driver的作用
- 4、ResourceManager 集群中的主节点 JobManager 中的负责资源管理的角色，和 TaskExecutor 一起构成资源管理的主从架构

当在任意地方发现要创建这四个组件的任何一个组件的实例对象的时候，创建成功了之后，都会要去执行他的 onStart()，在集群启动的源码分析中，其实这些组件的很多的工作流程，都被放在 onStart() 里面。

3.2. 阅读源码的准备

到底需要那些准备？

- 1、技术组件的大致工作原理
- 2、选择版本
- 3、搭建源码阅读的环境
- 4、场景驱动：脚本分析

3.3. Flink 集群启动脚本分析

Flink 集群的启动脚本在：flink-dist 子项目中，位于.flink-bin 下的 bin 目录：启动脚本为：start-cluster.sh

该脚本会首先调用 config.sh 来获取 masters 和 workers，masters 的信息，是从 conf/masters 配置文件中获取的，workers 是从 conf/workers 配置文件中获取的。然后分别：

- 1、通过 jobmanager.sh 来启动 JobManager
- 2、通过 taskmanager.sh 来启动 TaskManager

他们的内部，都通过.flink-daemon.sh 脚本来启动 JVM 进程，分析.flink-daemon.sh 脚本发现：

- 1、JobManager 的启动代号：standalonesession，实现类是：StandaloneSessionClusterEntrypoint
- 2、TaskManager 的启动代号：taskexecutor，实现类是：TaskManagerRunner

HDFS 集群一样的方式：

- 1、start-all.sh / start-dfs.sh
- 2、hadoop-daemon.sh start namenode/datanode/zkfc/journalnode
- 3、java org.apache.hadoop.server.namenode.NameNode

3.4. Flink 主节点 JobManager 启动分析

JobManager 是 Flink 集群的主节点，它包含三大重要的组件：

- 1、ResourceManager
Flink的集群资源管理器，只有一个，关于slot的管理和申请等工作，都由他负责
- 2、Dispatcher
负责接收用户提交的 JobGraph，然后启动一个 JobMaster，类似于 YARN 集群中的 AppMaster 角色，类似于 Spark Job 中的 Driver 角色
- 3、WebMonitorEndpoint
里面维护了很多很多的Handler，如果客户端通过 `flink run` 的方式来提交一个 job 到 flink 集群，最终，是由 `WebMonitorEndpoint` 来接收，并且决定使用哪一个 handler 来执行处理
例如：`submitJob ==> JobSubmitHandler`

当你提交一个 Job 到 Flink 集群运行的时候：

4、JobMaster/JobManager

负责一个具体的 Job 的执行，在一个集群中，可能会有多个 JobManager 同时执行，类似于 YARN 集群中的

AppMaster 角色，类似于 Spark Job 中的 Driver 角色
createJobManagerRunner()

关于 JobManager 的区分：

- 1、如果我们将 Flink 是主从架构，那么这个 JobManager 就是指主节点，它包含上面讲述的三种角色
- 2、如果我们将 Job 提交到 YARN 运行的时候，事实上，可以通过启动一个小集群的方式来运行，这个小集群的主节点也是 JobManager

你把job提交到 YARN 运行的时候，还有一种模式：job、session, Container (JobManager)
Container (StreamTask)

总结一下：

Flink 集群的主节点内部运行着：ResourceManager 和 Dispatcher，当 client 提交一个 job 到集群运行的时候（客户端会把该 Job 构建成一个 JobGraph 对象），Dispatcher 负责拉起 JobManager/JobMaster 来负责这个 Job 内部的 Task 的执行，执行Task所需要的资源，JobManager 向 ResourceManager 申请。

根据以上的启动脚本分析：JobManager的启动主类：StandaloneSessionClusterEntrypoint

```
// 入口
StandaloneSessionClusterEntrypoint.main()
    ClusterEntrypoint.runClusterEntrypoint(entrypoint);
        clusterEntrypoint.startCluster();
            runCluster(configuration, pluginManager);

// 第一步：初始化各种服务（7个服务）
initializeServices(configuration, pluginManager);

// 创建 DispatcherResourceManagerComponentFactory，初始化各种组件的
工厂实例

// 其实内部包含了三个重要的成员变量：
// 创建 ResourceManager 的工厂实例
// 创建 Dispatcher 的工厂实例
// 创建 WebMonitorEndpoint 的工厂实例
createDispatcherResourceManagerComponentFactory(configuration);

// 创建 集群运行需要的一些组件：Dispatcher， ResourceManager 等
// 创建 ResourceManager
// 创建 Dispatcher
// 创建 WebMonitorEndpoint
clusterComponent =
dispatcherResourceManagerComponentFactory.create(...)
```

第一步 initializeServices() 中做了很多服务组件的初始化：

```
// 初始化和启动 AkkaRpcService，内部其实包装了一个 ActorSystem
commonRpcService = AkkaRpcServiceUtils.createRemoteRpcService(...)

// 初始化一个负责 IO 的线程池
ioExecutor = Executors.newFixedThreadPool(...)
```

```
// 初始化 HA 服务组件，负责 HA 服务的是：ZooKeeperHaServices
haServices = createHaServices(configuration, ioExecutor);

// 初始化 BlobServer 服务端
blobServer = new BlobServer(configuration, haServices.createBlobStore());
blobServer.start();

// 初始化心跳服务组件，heartbeatServices = HeartbeatServices
heartbeatServices = createHeartbeatServices(configuration);

// 初始化一个用来存储 ExecutionGraph 的 store，实现是：
FileArchivedExecutionGraphStore
archivedExecutionGraphStore = createSerializableExecutionGraphStore(...)
```

第二步 createDispatcherResourceManagerComponentFactory(configuration) 中负责初始化了很多组件的工厂实例：

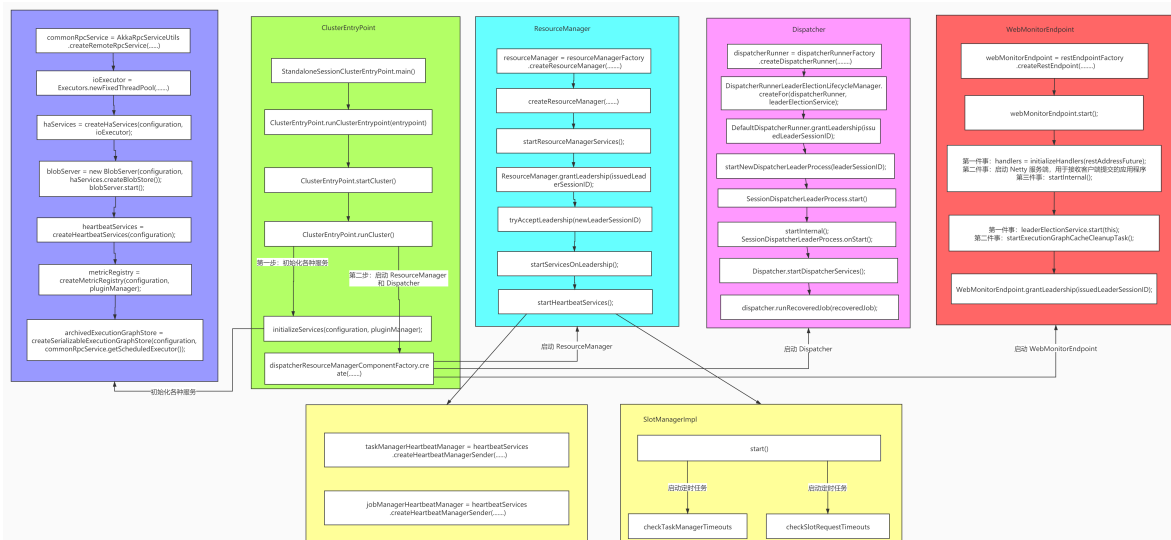
- 1、DispatcherRunnerFactory，默认实现：DefaultDispatcherRunnerFactory
- 2、ResourceManagerFactory，默认实现：StandaloneResourceManagerFactory
- 3、RestEndpointFactory，默认实现：SessionRestEndpointFactory

其中，DispatcherRunnerFactory 内部也实例化了一个：SessionDispatcherLeaderProcessFactory 组件

第三步 dispatcherResourceManagerComponentFactory.create(...) 中主要去创建 三个重要的组件：

- 1、DispatcherRunner，实现是：DefaultDispatcherRunner
- 2、ResourceManager，实现是：StandaloneResourceManager
- 3、WebMonitorEndpoint，实现是：DispatcherRestEndpoint

最终总结一下：



3.5. WebMonitorEndpoint 启动和初始化源码剖析

核心入口：

```
DispatcherResourceManagerComponentFactory.create(...)
```

第一件事：

```
// 初始化 webMonitorEndpoint
webMonitorEndpoint = restEndpointFactory.createRestEndpoint(
    configuration, dispatcherGatewayRetriever,
    resourceManagerGatewayRetriever,
    blobServer, executor, metricFetcher,

    highAvailabilityServices.getClusterRestEndpointLeaderElectionService(),
    fatalErrorHandler
);

// 启动 webMonitorEndpoint
webMonitorEndpoint.start();
```

3.6. ResourceManager 启动和初始化源码剖析

核心入口：

```
DispatcherResourceManagerComponentFactory.create(...)
```

第二件事：

```
// 初始化 ResourceManager
resourceManager = resourceManagerFactory.createResourceManager(
    configuration, ResourceID.generate(),
    rpcService, highAvailabilityServices, heartbeatServices,
    fatalErrorHandler, new ClusterInformation(hostname,
    blobServer.getPort()),
    webMonitorEndpoint.getRestBaseUrl(), metricRegistry,
    hostname
);

// 启动 ResourceManager
resourceManager.start();
```

3.7. Dispatcher 启动和初始化源码剖

核心入口：

```
DispatcherResourceManagerComponentFactory.create(...)
```

第二件事：

```
// 初始化 并启动 DispatcherRunner
dispatcherRunner = dispatcherRunnerFactory.createDispatcherRunner(

    highAvailabilityServices.getDispatcherLeaderElectionService(),
    fatalErrorHandler,

    // TODO_MA 注释： 注意第三个参数
    new
    HaServicesJobGraphStoreFactory(highAvailabilityServices),
    ioExecutor, rpcService, partialDispatcherServices
);
```

4. 本次课程总结

本次课程，主要讲解集群的启动, 今天主要讲解的是主节点 JobManager 的启动，在启动过程中，会有各种服务组件的初始化工作

- 1、Flink RPC 剖析
- 2、Flink 集群启动脚本分析
- 3、Flink 集群启动 JobManager 启动源码剖析
- 4、WebMonitorEndpoint 启动和初始化源码剖析
- 5、ResourceManager 启动和初始化源码剖析
- 6、Dispatcher 启动和初始化源码剖析

一定要注意，Flink Standalone 集群的主节点 JobManager 的内部包含非常重要的三大组件，在启动过程中，会依次启动，这三大组件分别是：

- 1、WebMonitorEndpoint
- 2、ResourceManager
- 3、Dispatcher

5. 本次课程作业

使用 Flink RPC 组件模拟实现 YARN，这个需求和我之前在讲 Spark 源码的时候，讲解的使用 Akka 模拟实现 YARN 的需求是类似的，只不过需要使用的技术是 Flink RPC 组件！

实现要求：

- 1、资源集群主节点叫做： **ResourceManager**，负责管理整个集群的资源
- 2、资源集群从节点叫做： **TaskExecutor**，负责提供资源
- 3、**ResourceManager** 启动的时候，要启动一个存活服务，制定一种机制（比如：某个 **TaskExecutor** 的连续5次心跳未接收到，则认为该节点死亡）实现下线处理
- 4、**TaskExecutor** 启动之后，需要向 **ResourceManager** 注册，待注册成功之后，执行资源（按照 Slot 进行抽象）汇报 和 维持跟主节点 **ResourceManager** 之间的心跳以便 **ResourceManager** 识别到 **TaskExecutor** 的存活状态